Progetto Algoritmi Paralleli e Sistemi Distribuiti: Parasites

Emanuele Conforti 220270

Contents

1 Introduzione

Il progetto in questione è un automa cellulare rappresentante un modello *preda-predatore*, in cui si hanno:

- **parasite**: rappresenta il predatore
- **vegetazione**: rappresenta la preda.

In sostanza, i parassiti attaccano e si cibano della vegetazione circostante, riproducendosi. Ove possibile, anche la vegetazione si riproduce e anche i parassiti possono morire, a causa di sovrappopolazione o mancanza di cibo.
All'interno del progetto è stata usata la libreria grafica **Allegro 5** per la stampa e visualizzazione dell'automa cellulare.

2 Struttura e implementazione

L'automa cellulare è stato rappresentato con un array lineare (*monodimensionale*) che simula una matrice (array a due dimensioni), in modo da avere i dati contigui in memoria. Per fare ciò, è stata implementata una funzione che converte gli indici di matrice in indici di array, nel seguente modo:

```
int coords(int r, int c) { return (r*COLS+c); }
```

Dove:

- r è l'indice di riga;
- c è l'indice di colonna;
- $COLS$ è il numero di colonne della matrice.

Il suddetto array è formato da celle di interi `int`. In questo modo è stato possibile rappresentare tutti i 5 stati che una cella può assumere (ogni cella avrà valore compreso tra 0 e 4):

- **EMTPY**: con valore pari a 0, è una cella vuota;
- **PARASITE**: con valore pari a 1, è la cella parassita (il predatore);
- **SEEDED_GRASS**: con valore uguale a 2, è il primo tipo di vegetazione, appena germogliata;
- **GROWING_GRASS**: con valore uguale a 3, è il secondo tipo di vegetazione in fase di crescita. Si tratta di una cella che alla precedente iterazione era allo stato SEEDED_GRASS;
- **GROWN_GRASS**: con valore uguale a 4, è l'ultimo tipo di vegetazione ormai matura ed il solo tipo di cui i parassiti possono cibarsi. Si tratta di una cella che alla precedente iterazione era allo stato GROWING_GRASS.

3 Configurazione iniziale

La matrice di partenza ha una sola cella PARASITE nella parte centrale, mentre le restanti celle sono tutte GROWN_GRASS. Così si da la possibilità all'unico predatore iniziale di potersi espandere e riprodurre in qualsiasi direzione.

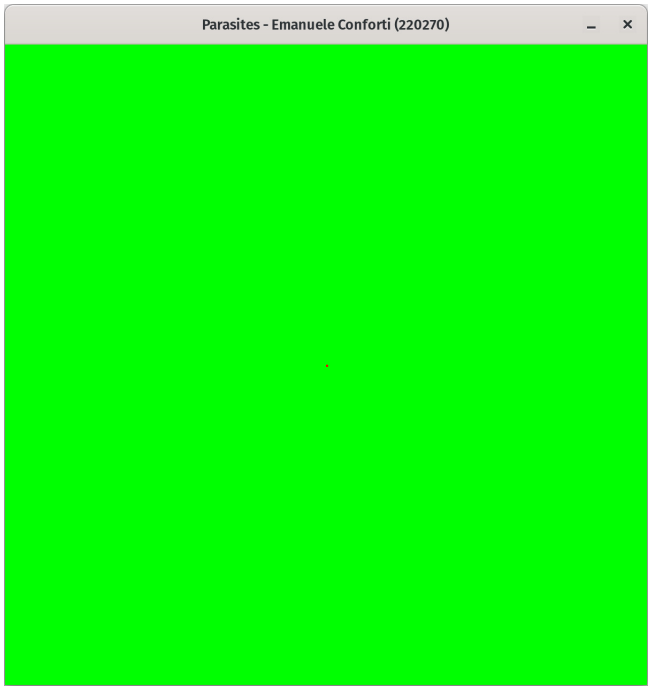

Figure 1: Snapshot del programma all'iterazione 1.

4 Funzione di transizione

Come ogni automa cellulare, anche questo ha una **funzione di transizione**, cioè una funzione che, data una cella con uno stato corrente, ne determina il suo stato successivo. Questa funzione viene applicata una volta per ogni iterazione, ad ogni cella dell'automa.
E' stato usato un meccanismo di turnificazione per l'applicazione della funzione di transizione sull'automa: in particolare, nelle iterazioni pari si può avere solo un passaggio agli stati SEEDED_GRASS, GROWING_GRASS, GROWN_GRASS e EMPTY. Invece, nelle iterazioni dispari, si ha il passaggio allo stato PARASITE.
Ciò significa che le celle prede si riprodurranno solo nelle iterazioni pari, mentre i parassiti si muoveranno solo nelle iterazioni dispari. In questo modo, l'ecosistema rappresentato si mantiene in equilibrio costante, senza la prevalenza

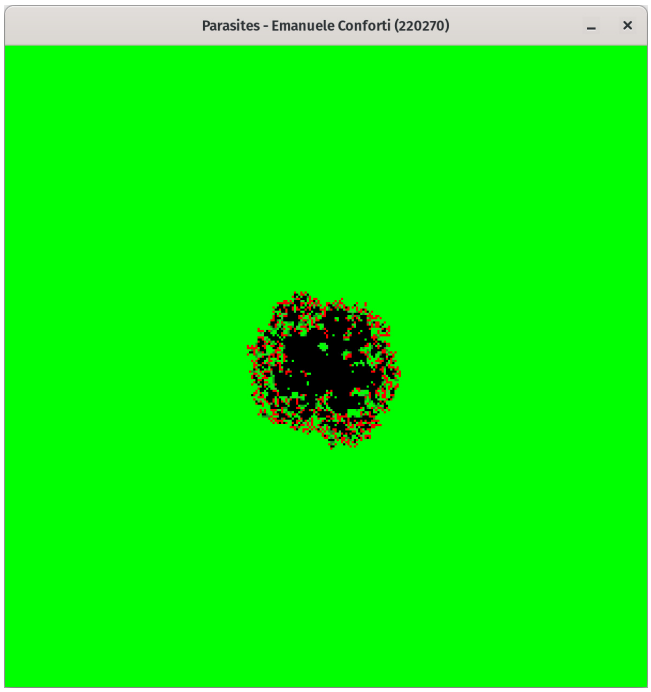

Figure 2: Snapshot del programma all'iterazione 100.

dei predatori sulle prede o viceversa.
La funzione di transizione prevede l'utilizzo del **vicinato di Moore** (che comprende tutte le 8 celle adiacenti alla cella corrente).

4.1 Iterazioni pari

Durante un'iterazione pari, una cella:

- EMPTY diventa SEEDED_GRASS se il numero di celle GROWN_GRASS adiacenti è ≥ 3, altrimenti rimane EMPTY;
- PARASITE rimane tale;
- SEEDED_GRASS diventa GROWING_GRASS;
- GROWING_GRASS diventa GROWN_GRASS;
- GROWN_GRASS rimane tale.

4.2 Iterazioni dispari

In questo caso, si sfrutta un numero *randNum* compreso tra 1 e 20 e generato in modo pseudo-casuale, attraverso l'utilizzo della funzione C++ `rand()` e di

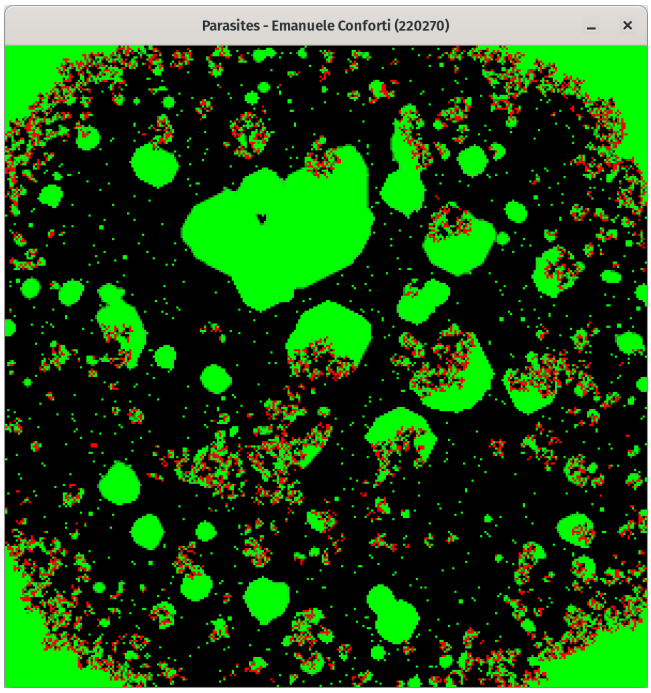

Figure 3: Snapshot del programma all'iterazione 500.

un *seed*.
Durante un'iterazione dispari, una cella:

- EMPTY rimane tale;
- PARASITE diventa EMPTY se il numero di celle PARASITE adiacenti è ≥ 5 (morte per sovrappopolazione) oppure se non ha alcuna cella GROWN_GRASS adiacente (morte per fame) oppure ancora se l'iterazione corrente è > 50 e *randNum* è ≤ 5 (morte casuale dopo la 50-esima iterazione con probabilità del 25%), altrimenti rimane PARASITE (continua a vivere);
- SEEDED_GRASS rimane tale;
- GROWING_GRASS rimane tale;
- GROWN_GRASS diventa PARASITE se ha almeno una cella PARASITE adiacente e se il *randNum* è ≤ 5, altrimenti rimane GROWN_GRASS. Così facendo, una cella GROWN_GRASS che ha un PARASITE vicino ha una probabilità del 25% di essere mangiato.

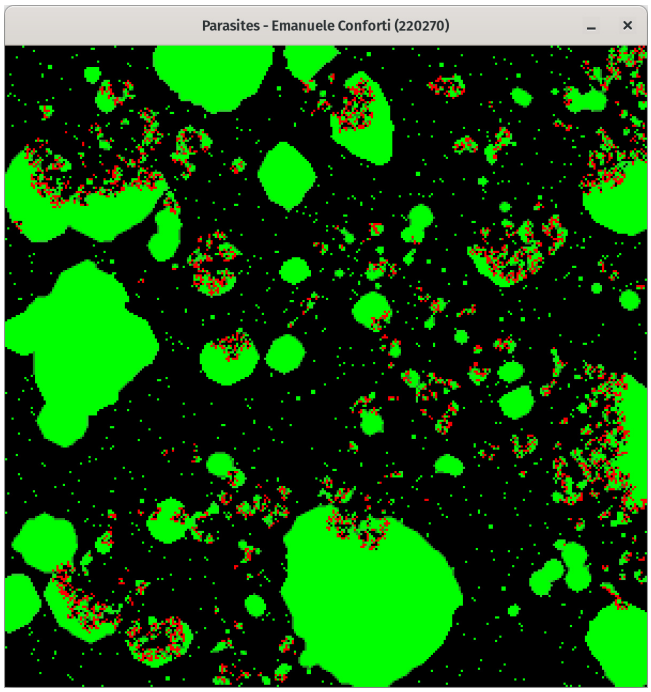

Figure 4: Snapshot del programma all'iterazione 1000.

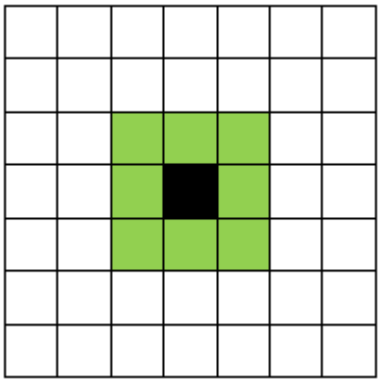

Figure 5: Struttura del vicinato di Moore.

5 Parallelizzazione e MPI

E' stata utilizzata la libreria MPI su linguaggio C++ per la parallelizzazione del programma, cioè per la sua esecuzione utilizzando più processi in parallelo. E' stata creata un'architettura **master-slave** dei processi: un processo, detto **master** si occupa della stampa a video dell'intero automa, mentre gli altri hanno

una sotto-matrice dell'automa ed eseguono su di essa la funzione di transizione. Nel caso di questo progetto, il processo con *rank*[1] 0 sarà il *master* (esegue la stampa a video) e si occuperà anche dello sviluppo di una parte dell'automa. Per la creazione dei processi sono state utilizzate le seguenti primitive MPI:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nthreads);
```

che servono rispettivamente a:

- definire il *rank* di ogni processo appartenente al comunicatore `MPI_COMM_WORLD`;
- definire il numero di processi *nthreads* da utilizzare.

La matrice viene poi divisa sui processi in modo da associare ad ognuno di essi una sotto-matrice `localMatrix` di dimensione $ROWS/nthreads * COLS$, dove:

- $ROWS$ è il numero di righe dell'automa;
- $COLS$ è il numero di colonne dell'automa;
- *nthreads* è il numero di processi usati.

In realtà, ogni processo avrà due sotto-matrici `localReadMatrix` e `localWriteMatrix`, rispettivamente per la lettura e scrittura sull'automa. Invece, il processo master avrà anche la matrice completa `matrix` da stampare ad ogni iterazione.
Per suddividere l'automa tra i processi, è stata creata una **topologia cartesiana ad una dimensione non toroidale**, attraverso le funzioni MPI:

```
MPI_Cart_create(MPI_COMM_WORLD, 1, dimensions, periods, 0, &comm);
MPI_Cart_shift(comm, 0, 1, &upNeighbor, &downNeighbor);
```

in cui:

- `MPI_Cart_create` crea un nuovo comunicatore *comm* a partire da `MPI_COMM_WORLD`. `dimensions` indica la dimensione della topologia (nel nostro caso, unidimensionale), invece `periods` specifica la **toroidalità** della matrice, cioè la possibilità che la riga 0 sia adiacente alla riga *ROWS-1* e viceversa;
- `MPI_Cart_shift` assegna ad ogni processo due processi *vicini* (con i quali comunicherà), per ogni dimensione della topologia.

Ogni processo ha bisogno di scambiare le **celle halo** con i processi adiacenti, cioè celle adiacenti alle celle ai bordi di `localMatrix`, ma che fanno parte della sotto-matrice gestita da un altro processo. Dunque, la dimensione delle `localMatrix` è stata ampliata aggiungendo due nuove righe e arrivando ad essere uguale a $(ROWS/nthreads + 2) * COLS$, per poter ospitare le celle halo, che sarebbero l'ultima riga del processo precedente e la prima riga di quello successivo. Ciò non vale per il primo e l'ultimo processo, i quali hanno solo una riga di celle halo, non essendovi toroidalità. Per lo scambio di celle halo tra processi, è stato inizializzato un nuovo tipo di dato `borderType` composto da $COLS$ (numero di colonne dell'automa) interi contigui, nel seguente modo:

[1] identificativo di un processo

```
MPI_Type_contiguous(COLS, MPI_INT, &borderType);
```

Inoltre, alla fine di ogni iterazione, ogni processo dovrà anche inviare la propria sotto-matrice locale al processo 0, che si occuperà della stampa. Perciò, è stato implementato il tipo di dato `localMatrixType`, composto da *(ROWS/nthreads)*COLS* (dimensione di `localReadMatrix` senza celle halo) interi contigui, nel seguente modo:

```
MPI_Type_contiguous((ROWS/nthreads)*COLS, MPI_INT, &localMatrixType);
```

Dopo aver fatto ciò, inizia la vera e propria esecuzione dell'automa cellulare:

1. `MPI_sendBorders()`: ogni processo invia i propri bordi (prima e ultima riga di `localReadMatrix`) in modo **asincrono** o **non bloccante**, cioè senza attendere che il destinatario riceva i dati;
2. `transFunctionInside()`: ogni processo applica la funzione di transizione alle celle di `localReadMatrix` che non hanno celle halo come adiacenti;
3. `MPI_recvBorders()`: ogni processo riceve le celle halo dai processi vicini;
4. `transFunctionBorders()`: ogni processo applica la funzione di transizione solo alle celle che hanno celle halo come adiacenti;
5. `swap()`: ogni processo inverte la propria `localReadMatrix` con la `localWriteMatrix`;
6. `MPI_Gather`: ogni processo invia al processo master la propria `localReadMatrix` per la stampa;
7. `print()`: il processo master stampa a video `matrix`;
8. `MPI_Bcast`: il processo master invia a tutti gli altri le variabili `GEN`, che rappresenta il numero dell'iterazione corrente e `end`, che indica la fine dell'esecuzione (1 se bisogna terminare, 0 altrimenti).

5.1 Scambio dei bordi

Grazie alla funzione `MPI_Cart_shift(comm, 0, 1, &upNeighbor, &downNeighbor)`, ogni processo conosce i suoi processi adiacenti, ovvero `upNeighbor` (il precedente) e `downNeighbor` (il successivo). Da tenere a mente che il primo processo ha solo il vicino `downNeighbor`, mentre l'ultimo processo ha solo `upNeighbor`.
Per l'invio dei bordi, è stata utilizzata la primitiva MPI `MPI_Isend()`: si tratta della versione *non bloccante* della funzione `MPI_Send()`. E' stato implementato un invio dei bordi asincrono per via dell'aumento delle performance del programma: infatti, in questo modo i processi mittente continuano a lavorare senza attendere al ricezione da parte dei destinatari.
Tutti i processi inviano la prima e l'ultima riga di `localReadMatrix` (`localReadMatrix[1]` e `localReadMatrix[ROWS/nthreads]`, perchè in `localReadMatrix[0]` e `localReadMatrix[ROWS/nthreads+1]` sono contenute le celle halo), eccetto il primo e l'ultimo processo che inviano solo una riga, rispettivamente l'ultima e

la prima.
La ricezione dei bordi è stata svolta tramite la primitiva MPI `MPI_Recv()`. Anche in questo caso, il primo e l'ultimo processo hanno ricevuto una solo riga, avendo un solo vicino.

6 Misure e tempi

Le misurazioni sono state eseguite sulla seguente architettura:
CPU Intel Core i7-9750H @ 2.60GHz, *16,0 GB RAM*, *GPU Nvidia GeForce GTX 1650*. I principali valori calcolati sono stati:

- **tempo di esecuzione**;
- **speedup**: indica il rapporto tra il tempo di esecuzione seriale e il tempo di esecuzione parallelo;
- **efficienza**: indica il rapporto tra speedup e numero di processori usati.

I tempi sono stati presi in base ai seguenti criteri:

- *numero di threads*: il programma è stato eseguito con 1, 2, 4, 6 e 8 threads;
- *dimensione* della matrice: sono state scelte le dimensioni 120x120, 360x360 e 720x720;
- *steps*, ovvero numero di iterazioni compiute: le misure sono state prese sulle iterazioni 100 e 1000.

```
- STEPS: 100

--- Dimensione matrice: 120x120 ---
    1 thread  --- Time:   40,817449   Speedup: 1         Efficienza: 1
    2 threads --- Time:   22,364789   Speedup: 1,825076  Efficienza: 0,912538
    4 threads --- Time:   12,904173   Speedup: 3,163120  Efficienza: 0,790780
    6 threads --- Time:    9,639972   Speedup: 4,234187  Efficienza: 0,705697
    8 threads --- Time:   12,526796   Speedup: 3,258410  Efficienza: 0,407301

--- Dimensione matrice: 360x360 ---
    1 thread  --- Time:   330,461872   Speedup: 1         Efficienza: 1
    2 threads --- Time:   172,147691   Speedup: 1,919641  Efficienza: 0,959820
    4 threads --- Time:    92,233505   Speedup: 3,582883  Efficienza: 0,895720
    6 threads --- Time:    80,432903   Speedup: 4,108541  Efficienza: 0,684756
    8 threads --- Time:    84,661510   Speedup: 3,903330  Efficienza: 0,487916

--- Dimensione matrice: 720x720 ---
    1 thread  --- Time:  1305,485751   Speedup: 1         Efficienza: 1
    2 threads --- Time:   679,724381   Speedup: 1,920610  Efficienza: 0,960305
    4 threads --- Time:   359,474498   Speedup: 3,631650  Efficienza: 0,907912
```

```
    6 threads --- Time:   265,026302   Speedup: 4,925872  Efficienza: 0,820979
    8 threads --- Time:   343,065548   Speedup: 3,805353  Efficienza: 0,475669


- STEPS: 1000

--- Dimensione matrice: 120x120 ---
    1 thread  --- Time:   484,803521   Speedup: 1         Efficienza: 1
    2 threads --- Time:   255,254316   Speedup: 1,899296  Efficienza: 0,949648
    4 threads --- Time:   141,719094   Speedup: 3,420876  Efficienza: 0,855219
    6 threads --- Time:   110,953586   Speedup: 4,369426  Efficienza: 0,728237
    8 threads --- Time:   131,186660   Speedup: 3,695524  Efficienza: 0,461940

--- Dimensione matrice: 360x360 ---
    1 thread  --- Time:  4133,523351   Speedup: 1         Efficienza: 1
    2 threads --- Time:  2221,063300   Speedup: 1,861056  Efficienza: 0,930528
    4 threads --- Time:  1234,553059   Speedup: 3,348194  Efficienza: 0,837048
    6 threads --- Time:   881,437953   Speedup: 4,689522  Efficienza: 0,781587
    8 threads --- Time:  1159,343827   Speedup: 3,565399  Efficienza: 0,445674

--- Dimensione matrice: 720x720 ---
    1 thread  --- Time: 15238,513579   Speedup: 1         Efficienza: 1
    2 threads --- Time:  8041,688153   Speedup: 1,894939  Efficienza: 0,947470
    4 threads --- Time:  4408,507069   Speedup: 3,456615  Efficienza: 0,864153
    6 threads --- Time:  3254,264158   Speedup: 4,682629  Efficienza: 0,780438
    8 threads --- Time:  4116,876140   Speedup: 3,701474  Efficienza: 0,462684
```

6.1 Considerazioni

Si può notare quasi sempre un dimezzamento del tempo di esecuzione quando viene raddoppiato il numero di threads usati, fino all'utilizzo di 4 threads. Per quanto riguarda l'uso di 6 e 8 threads, il tempo di esecuzione parallela aumenta, a causa del forte **overhead**, cioè il lavoro extra che ogni processo dovrà eseguire, come comunicazione con altri processi oppure inizializzazione e terminazione di processi.
Allo stesso modo, vi è anche un incremento nello speedup finchè si utilizza un numero di threads ≤ 6. Quando se ne usano 8, ancora una volta le performance si riducono per l'overhead.
Invece l'efficienza, tranne nell'utilizzo di 8 threads, si mantiene quasi costante, soprattutto nelle esecuzioni con stesso numero di threads, ma diversa dimensione della matrice (quindi, dimensione di dati del programma). Infatti, in alcuni casi, mantenendo fisso il numero di threads usati, all'aumentare della dimensione della matrice, l'efficienza non cambia: ciò è dovuto alla **funzione di isoefficienza**, in base alla quale mantenendo costante il rapporto tra dimensione del programma (in questo caso, in termini di dati usati) e overhead totale

parallelo, rimarrà costante anche l'efficienza.